Chapter 5

Built-In Functions

Problem 5.1

Each of the following Scheme expressions denotes/represents some kind of Scheme datum. For each, state the data type *(e.g., number, boolean, symbol, list, function, etc.) of the Scheme datum it represents. In addition, specify the data type of the Scheme datum it* evaluates *to. The first one is done for you as an illustration.*

Expression	*Represents a datum of this type*	*Evaluates to a datum of this type*
`#t`	`boolean`	`boolean`
`+`		
`/`		
`()`		
`84`		
`3/5`		
`void`		

Chapter 6

Non-Empty Lists

Problem 6.1

Each of the following Scheme expressions denotes/represents some kind of Scheme datum. For each, state the data type *(e.g., number, boolean, symbol, list, function, etc.) of the Scheme datum it represents. In addition, specify the data type of the Scheme datum it* evaluates *to. The first one is done for you as an illustration.*

Expression	*Represents a datum of this type*	*Evaluates to a datum of this type*
`#t`	`boolean`	`boolean`
`(* 4 6)`		
`eval`		
`(eval 3)`		
`(void)`		

Problem 6.2

Describe in detail the steps that DrScheme goes through in evaluating the expression `(void)`. *Be sure to carefully distinguish the* void *datum and the built-in* `void` *function.*

Chapter 7

Special Forms

Problem 7.1

When the Default Rule is used to evaluate a non-empty list, the first step is to evaluate each element in the list. However, special forms are not *evaluated by the Default Rule. As a result, it can happen that some of the elements in a special form* are *evaluated, while others are* not. *For this problem, summarize the following information about the* `define` *and* `quote` *special forms:*

(1) how many elements there are—aside from the keyword symbol;

(2) which elements get evaluated and which do not;

(3) whether there is an output value and, if so, how it is computed; and

(4) whether there is a side effect and, if so, what it is.

Problem 7.2

For each statement below, decide which of the words in parentheses apply:

- *Evaluation of a* `define` *special form (always, never, sometimes) causes a side effect.*
- *Evaluation of a* `quote` *special form (always, never, sometimes) causes a side effect.*

Chapter 8

Predicates

Problem 8.1

Each of the following Scheme expressions denotes/represents some kind of Scheme datum. For each, state the data type *(e.g., number, boolean, symbol, list, function, etc.) of the Scheme datum it represents. In addition, specify the data type of the Scheme datum it* evaluates *to. The first one is done for you as an illustration.*

Expression	*Represents a datum of this type*	*Evaluates to a datum of this type*
`(* 4 6)`	*list*	*number*
`'cs101`		
`(> 4 3)`		
`(number? 'x)`		
`(symbol? 'x)`		
`(symbol? eval)`		

Problem 8.2

Write down a contract for the built-in >= function.

Problem 8.3

Explain the steps taken by DrScheme in the following interactions:

```
> +
#<procedure:+>
> (define addn +)
> (addn 4 5)
9
> (define function? procedure?)
> (function? +)
#t
```

Problem 8.4

Explain the output values generated by the following sequence of interactions.

```
> (define myvar 'sunday)
> (define yourvar 'monday)
> (eq? myvar 'sunday)
#t
> (eq? myvar myvar)
#t
> (eq? myvar 'myvar)
#f
> (eq? myvar yourvar)
#f
> (eq? yourvar 'monday)
#t
> (eq? yourvar 'sunday)
#f
```

Problem 8.5

Describe in detail the steps that DrScheme goes through in evaluating the expression `(void? (void))`. *Be sure to carefully distinguish the* void *datum, the built-in* `void` *function, and the built-in* `void?` *type-checker predicate.*

Chapter 9

Defining Functions

Problem 9.1

Consider the following Interactions Window session:

```
    > (define pie 3.14159)
    > (define funk (lambda (x) (* x pie)))
    > (funk 10)
    31.4159
```

Accurately describe the process that DrScheme goes through in evaluating these three expressions to generate the result, `31.4159`*. Strive to be complete, while also being concise.*

Chapter 10

Some practicalities

Problem 10.1

Write down a contract for the tester function using the form below:

Name:
Input:
Output:
Side Effects:

Problem 10.2

DrScheme uses the Default Rule to evaluate the list denoted by `(tester '(+ 1 2))`. *As indicated above, the result of evaluating this list is the number* 3. *Carefully describe the process DrScheme goes through to generate this result. (You may wish to review Examples 9.3.3 and 9.3.4 to recall how DrScheme applies a function to inputs.) In particular, what value is associated with the input parameter* `datum` *in the local environment? What value is passed to the* `printf` *function called in the body of the* `tester` *function? And what steps does DrScheme go through to evaluate the expression* `(eval datum)` *in the body of the* `tester` *function?*

Problem 10.3

How would you change the definition of the `tester` *function so that it* printed out *the result of evaluating the Scheme datum instead of returning it as the output value? In this case, the* `tester` *function would return the "no value" datum.*

Chapter 11

Conditional Expressions I

Problem 11.1: The `maxx` function

Define a function, called `maxx`*, that takes two numbers as its inputs. It should return the maximum of the two numbers, as illustrated below:*

```
> (maxx 2 3)
3
> (maxx 5 1)
5
> (maxx 4 4)
4
```

As you may have guessed, there is a built-in `max` *function. However, you are not allowed to use it for this problem! Instead, use* `if` *to determine which input number is bigger. The operation of the* `maxx` *function could be described thusly: if* `x` *is bigger than* `y`*, then the output should be* `x`*; otherwise, it should be* `y`*. Here's the contract:*

```
;;  MAXX
;; -------------------------------------------
;;  INPUTS:  X, Y, two numbers
;;  OUTPUT:  The maximum of X and Y
```

And some `tester` *expressions:*

```
(tester '(maxx 2 3))
(tester '(maxx 5 1))
(tester '(maxx 4 4))
```

Problem 11.2: Printing out a message about bananas!

Here's the contract for the `banana-msg` *function. Note that it does* not *generate any* output value; *however, it* does *cause some* side-effect printing *to occur in the Interactions Window:*

```
;;  BANANA-MSG
;; -----------------------------
;;  INPUT:  NUM, an integer
;;  OUTPUT:  don't care
;;  SIDE EFFECT:  Print out a message in the Interactions Window
;;    such as:  I ate NUM bananas!, except that NUM should be
;;    replaced by its value.  Also, if you only ate one banana,
;;    then banana should not be pluralized!
```

Here are some examples of its use (in the Interactions Window):

```
> (banana-msg 3)
I ate 3 bananas!            ⟵  these are not output values!!
> (banana-msg 1)
I ate 1 banana!             ⟵  they are side-effect printing!!
> (banana-msg 0)
I ate 0 bananas!
```

Here are some `tester` *expressions to copy into your Definitions Window:*

```
(tester '(banana-msg -2))
(tester '(banana-msg 0))
(tester '(banana-msg 1))
(tester '(banana-msg 3))
```

There are many ways to solve this problem. Recall the description of the `printf` *function in Chapter 10.*

Problem 11.3

Define a function, called `quadrant`*, that satisfies the following contract:*

```
;;  QUADRANT
;; ---------------------------
;;  INPUTS:  X, Y, two numbers
;;  OUTPUT:  A number specifying the quadrant to which the point
;;    (X,Y) belongs in the XY-plane; or 0 if it lies on an axis.
```

Recall that the first quadrant is where both x and y are positive; the second quadrant is where x is negative and y is positive; the third quadrant is where both x and y are negative; and the fourth quadrant is where x is positive and y is negative. Be sure to test your function on a variety of inputs (for all four quadrants and various places on the axes, including the origin).

Problem 11.4

Define a function, called `data-type-of`*, that satisfies the following contract. Note that it returns a* symbol *as its output.*

```
;;  DATA-TYPE-OF
;; ----------------------------------
;;  INPUT:   DATUM, anything
;;  OUTPUT:  A SYMBOL representing the data type of DATUM,
;;    one of:  NUMBER, BOOLEAN, LIST, SYMBOL, STRING, etc.
```

Note: You don't need to handle every possible data type. Returning the symbol `unknown` *is okay if you get tired. Here are some examples of its behavior in the Interactions Window:*

```
> (data-type-of #t)
boolean
> (data-type-of ())
list
> (data-type-of 45)
number
```

And here are some `tester` *expressions to copy into your Definitions Window:*

```
(tester '(data-type-of 3))
(tester '(data-type-of #t))
(tester '(data-type-of '(+ 2 3)))
(tester '(data-type-of (+ 2 3)))
(tester '(data-type-of "abc"))
```

Hint: Use the built-in type-checker predicates from Chapter 8.

Problem 11.5

The `implies` *function takes two boolean inputs, and generates a boolean output, as illustrated below:*

```
(implies #t #t) ===> #t
(implies #f #t) ===> #t
(implies #t #f) ===> #f
(implies #f #f) ===> #t
```

Write a contract for the `implies` *function, and then implement it in Scheme.*

⋆ *There are only four different combinations of inputs for this function; however, you can include more complicated input expressions, such as:* `(implies (> 3 2) (< 4 5))`. *And since anything other than* `#f` *counts as true, you can even do things like:* `(implies 'hi 'there)`.

Chapter 12

Recursion I

Problem 12.1

Define a function, called `power`*, that takes two inputs:* x*, any real number, and* p *any non-negative integer. It should return as its output the value of* x *raised to the* p^{th} *power (i.e.,* x^p*), as illustrated below.*

```
> (power 2 3)            ⟵ 2³ = 2·2·2 = 8
8
> (power 3 2)            ⟵ 3² = 3·3 = 9
9
> (power 2 5)            ⟵ 2⁵ = 2·2·2·2·2 = 32
32
```

- ⋆ *Hint: Use recursion, similar to how it is used in* `facty-v1` *from Example 12.1.2. For example, note that* $2^9 = 2 \cdot (2^8)$.
- ⋆ *Be sure to include a contract for your function.*

Chapter 13

Conditional Expressions II

Problem 13.1

For each statement below, decide which of the words in parentheses apply:

- *Evaluation of an* `if` *special form (always, never, sometimes) causes a side effect.*
- *Evaluation of an* `and` *special form (always, never, sometimes) causes a side effect.*
- *Evaluation of an* `or` *special form (always, never, sometimes) causes a side effect.*
- *Evaluation of an* `if` *special form always requires evaluating* exactly *(one, two, all) of its inputs.*
- *Evaluation of an* `and` *special form always requires evaluating* at least *(one, two, all) of its inputs.*
- *Evaluation of an* `or` *special form always requires evaluating* at least *(one, two, all) of its inputs.*

Problem 13.2

Recall that times in the 24-hour military clock involve hours that range from 0 to 23. For example, 00:00 corresponds to midnight; 08:23 is sometime in the morning; 12:00 corresponds to noon; and 15:39 is sometime in the afternoon.

(a) *Define a function, called* `time-of-day`*, that takes two numerical inputs,* `mil-hours` *and* `minutes`*, where* `mil-hours` *represents the number of hours according to the 24-hour military clock, and* `minutes` *represents the number of minutes.*

⋆ *You may assume that* `mil-hours` *and* `minutes` *are integers such that:*
$0 \leq \mathtt{mil-hours} < 24$ *and* $0 \leq \mathtt{minutes} < 60$.

⋆ `time-of-day` *should return a* symbol *as its output. In particular, it should return one of the following:* `midnight`, `am`, `noon` *or* `pm`*, as appropriate. For example:*

```
> (time-of-day 15 39)
pm
> (time-of-day 12 0)
noon
```

Note that the `pm` *and* `noon` *are output values that are symbols; they are* not *side-effect printing.*

⋆ *HINT: You may use* `if` *or* `cond` *in the body of your function. In either case, be sure to include comments that briefly describe each case that you're handling.*

- ⋆ *NOTE: 12:00 midnight is neither a.m. nor p.m. Rather, it is a boundary between a.m. and p.m. Similar remarks apply to 12:00 noon. So* `(time-of-day 12 0)` *should return* `noon`*, not* `am` *or* `pm`*. Similarly* `(time-of-day 0 0)` *should return* `midnight`*, not* `am` *or* `pm`*. But* `(time-of-day 0 15)` *should return* `am`*, since 00:15 in military time corresponds to 12:15 a.m. in civilian time.*
- ⋆ *Be sure to write a contract for your function!*

(b) Define a function, called `mil-to-civil-hrs`*, that takes a single numerical input,* `mil-hours`*, where* $0 \leq \texttt{mil} - \texttt{hours} < 24$*. It should generate as its output, the corresponding number of hours according to the 12-hour civilian clock. For example:*

```
(mil-to-civil-hrs 19) ===> 7
```

Because 19:00 on the military clock corresponds to 7:00 p.m. on the civilian clock.

Hint: Use a `cond` *in the body of your function.*

Hint: Be careful about 0*.*

Be sure to include a contract for your function!

(c) Define a function, called `print-civil-time-from-mil`*. It should take two numerical inputs,* `mil-hours` *and* `minutes`*, as in part (a). However, this function, unlike the above functions, should* not *generate any Scheme datum as output. Instead, it should have the* side effect *of displaying the time in the 12-hour civilian format, as the following interactions window session illustrates:*

```
> (print-civil-time-from-mil 15 39)
3:39 pm
```

- ⋆ *In this example,* `3:39 pm` *is side-effect printing, generated using the built-in* `printf` *function. There is no* output *value.*
- ⋆ *Use the* `time-of-day` *and* `mil-to-civil-hrs` *functions as helpers. You shouldn't need to re-implement the computations done by* `time-of-day` *or* `mil-to-civil-hrs`*.*
- ⋆ *Be sure to include a contract for your function!*

(Optional) If the number of minutes is small, you will have to work a little harder to make sure that a leading zero *is displayed. Consider* `3:6 pm` *versus* `3:06 pm`*.*

```
> (print-civil-time-from-mil 15 6)
3:06 pm
```

There are many examples that demonstrate the use of the built-in `printf` *function in the* `amst-helper.txt` *file available in each lab and assignment directory.*

Problem 13.3

Define a function, `compute-tax`*, that takes a single input,* `income`*. It should return the amount of tax owed on that income, as determined by the following* tax brackets:

Income below $10,000 is taxed at 10%.

Income between *$10,000 and $30,000 (only the amount* after *the first $10,000) is taxed at 15%.*

Income above *$30,000 (only the amount* after *the first $30,000) is taxed at 25%.*

$\Longrightarrow$ *If you make more than $10,000, the first $10,000 of your income is taxed at 10%; only the amount above $10,000 is taxed at the higher rates. Similarly, if you make more than $30,000, the first*

$10,000 of income is taxed at 10%, the next $20,000 (i.e., the amount between $10,000 and $30,000) is taxed at 15%, and the rest is taxed at 25%. Thus, if you earn $100,000, your taxes will not *be $25,000 (i.e., 25% of $100,000), instead, they will be:*

(10% of $10,000) + (15% of $20,000) + (25% of $70,000)

which equals: $1,000 + $3,000 + $17,500 = $21,500.

Be sure to include `tester` *expressions that test a representative set of cases. You may use* `if` *or* `cond` *for this problem. And, as always, be sure to include a contract for your function.*

Problem 13.4

The CMPU-101 bookstore is open on Saturdays from 11:45 a.m. to 12:15 p.m., inclusive, and all day Tuesday, except *from 12:01 p.m. to 12:59 p.m., inclusive.*

(a) *Define a function, called* `bookstore-open?`, *that satisfies the following contract. Use a* `cond` *special form to structure the body of this function.*

```
;;  BOOKSTORE-OPEN?
;; ---------------------------------------
;;  INPUTS:  DAY, a symbol, one of SUN, MON, TUE, ..., FRI, SAT
;;           HOUR, an integer from 1 to 12, inclusive
;;           MINUTES, an integer from 0 to 59, inclusive
;;           AM-OR-PM, a symbol, either AM or PM
;;  OUTPUT:  #t, if the inputs specify one of the following:
;;    Saturday from 11:45 am to 12:15 pm, inclusive.
;;    All day Tuesday, except from 12:01 pm to 12:59 pm,
;;    inclusive; #f otherwise.
```

Here are some examples of its behavior:

```
> (bookstore-open? 'sat 11 30 'am)
#f
> (bookstore-open? 'sat 11 50 'am)
#t
> (bookstore-open? 'sat 12 12 'pm)
#t
> (bookstore-open? 'sat 12 49 'pm)
#f
```

(b) *Same as above, except this time use the boolean operators,* `and`, `or` *and* `not`, *instead of conditional expressions, as described in Section 13.3.*

Problem 13.5

The following predicate is defined using a `cond` *special form:*

```
(define office-open?
  (lambda (day am-or-pm)
    (cond
     ;; Case 1:  Closed on Fridays
```

```
        ((eq? day 'fri)
         #f)
        ;; Case 2:  Open Wed afternoons
        ((and (eq? day 'wed)
              (eq? am-or-pm 'pm))
         #t)
        ;; Case 3:  Closed on Tuesday mornings
        ((and (eq? day 'tue)
              (eq? am-or-pm 'am))
         #f)
        ;; Case 4:  Open all other times
        (else
         #t))))
```

Your job is to define a predicate, called `office-open?-alt`*, that works just like* `office-open?`*, except that it is defined using* `and`*,* `or` *and* `not`*, instead of the conditional expressions,* `if` *or* `cond`*.*

⋆ *Careful! Some of the cases above output* `#t`*, while others output* `#f`*.*

Chapter 14

Recursion II

Problem 14.1

Define a function, called `sum-recips`*, that computes sums of the following form:*

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

where `n` *is some positive integer. Here are some sample interactions:*

```
> (sum-recips 1)
1
> (sum-recips 2)
3/2
> (sum-recips 3)
11/6
```

And the corresponding `tester` *expressions:*

```
(tester '(sum-recips 1))
(tester '(sum-recips 2))
(tester '(sum-recips 3))
```

Insert some more `tester` *expressions of your own. If you want to encourage DrScheme to display numbers in "floating point" form (e.g.,* `1.5` *instead of* `3/2`*), just use* `1.0` *in your base case, instead of* `1`*. Consider the following:*

```
> (/ 3 2)
3/2
> (/ 3.0 2)
1.5
```

Be sure to include a contract for your function!

Problem 14.2

Define a function, called `alt-sum`*, that takes a positive integer* `n` *as its only input. It should return as its output the following sum:*

$$1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \ldots \pm \frac{1}{n}$$

where the sign of each term is negative if the denominator is even, and positive if the denominator is odd. Here are some examples:

```
> (alt-sum 1)
1
> (alt-sum 2)
0.5
> (alt-sum 3)
0.8333333333333333
```

For example, `(alt-sum 3)` *returns the sum,* $1 - \frac{1}{2} + \frac{1}{3} = \frac{5}{6}$. *To ensure that you get the desired "floating point" notation, you can use expressions such as* `(/ 1.0 n)` *instead of* `(/ 1 n)`, *as illustrated below:*

```
> (/ 1 5)
1/5
> (/ 1.0 5)
0.2
```

This is especially relevant for cases where `n` *is large. The value of* `(alt-sum 100)` *in fractional notation would be very cumbersome!*

⋆ *There are built-in functions called* `even?` *and* `odd?` *that return* `#t` *if their input is an even (or odd) number, as illustrated below.*

```
> (even? 4)
#t
> (even? 9)
#f
> (odd? 4)
#f
> (odd? 9)
#t
```

The following set of problems involve functions that do not generate any output value, but instead cause side-effect printing to occur. For such functions, the following `alt-tester` function will generate nicer looking test results in the Interactions Window.

```
;;  ALT-TESTER
;; -------------------------------------
;;  INPUT:  DATUM, anything
;;  OUTPUT: void
;;  SIDE EFFECT:  Displays DATUM, then prints a newline,
;;    then evaluates DATUM, and finally prints another newline.

(define alt-tester
  (lambda (datum)
    (printf "~A ==>" datum)
    (newline)
    (eval datum)
    (newline)))
```

It is the same as the `tester` function seen earlier, except that it makes sure that any side-effect printing caused by evaluating the expression `(eval datum)` starts on a new line, and it does not generate any output value! To enable use of this function, copy-and-paste the above definition into your Definitions Window.

Problem 14.3

Why would it be difficult to implement the `print-n-dashes` *function from Example 4.2.1 using* `if` *instead of* `cond`*?*

Problem 14.4

Define a function, called `print-thing-n-times`*, that takes two inputs:* `thing` *and* `n`*, where* `thing` *can be anything, and* `n` *is a non-negative integer. It should* not *generate an output value; instead, it should have the* side effect *of* printing out `thing` `n` *times in the Interactions Window, as illustrated below:*

```
> (print-thing-n-times 'Hi 5)
HiHiHiHiHi
> (print-thing-n-times '-*- 3)
-*--*--*-
```

Be sure to include a contract for your function.

Problem 14.5

Define a function, called `print-down-to-zero`*, that takes a non-negative integer* `n` *as its only input. It should* not *generate any output value; instead, it should have the* side effect *of printing out the values from* `n` *down to zero in the Interactions Window, as illutrated below.*

```
> (print-down-to-zero 5)
5 4 3 2 1 0
> (print-down-to-zero 22)
22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

Be sure to include a contract for your function.

Problem 14.6: Printing rectangles and squares

Copy-and-paste the contract and function definition for the `print-n-dashes` *function from Example 4.2.1 into your Definitions Window. Recall that* `print-n-dashes` *does* not *generate any Scheme output value; instead, it has the* side effect *of displaying a row of* `n` *dashes in the Interactions Window, as illustrated below.*

```
> (print-n-dashes 5)
-----
> (print-n-dashes 12)
------------
```

(Printing Rectangles) *For this part, you must define a function called* `print-rectangle`*, that takes two inputs, both of which are non-negative integers. This function should not generate any output value. Instead, it should have the following side effect: It should display a rectangular pattern of dashes whose number of rows and number of columns correspond to the two numerical inputs, as illustrated below.*

```
> (print-rectangle 5 2)
--
--
--
--
--
> (print-rectangle 2 5)
-----
-----
```

Here are some hints:

- *Pay attention to which input specifies the number of rows, and which specifies the number of columns.*
- *Use recursion.*
- *Use* `print-n-dashes` *as a helper function to print individual rows.*

(Printing Squares) *Define a function, called* `print-square`*, that takes a single non-negative integer as its only input. It should not generate any output value, but instead should print out a square pattern of dashes in the Interactions Window, whose number of rows and columns is specified by the single numerical input, as illustrated below.*

```
> (print-square 3)
---
---
---
> (print-square 4)
----
----
----
----
```

Hint: Use the `print-rectangle` *function as a helper. Your* `print-square` *function should not be complicated!*

Problem 14.7: Printing upside-down triangles

Copy-and-paste the contract and definition for the `print-n-dashes` *function from Example 4.2.1 into your Definitions Window. Then define a function, called* `print-upside-down-triangle`*, that satisfies the following contract:*

```
;;  PRINT-UPSIDE-DOWN-TRIANGLE
;; -------------------------------------
;;  INPUT:   NUM-ROWS, a non-negative integer
;;  OUTPUT:  Nothing
;;  SIDE EFFECT:  Prints an upside-down triangle in the
;;   Interactions Window consisting of NUM-ROWS rows.
```

Here are some examples of its behavior:

```
> (print-upside-down-triangle 3)
---
```

```
--
-

> (print-upside-down-triangle 5)
-----
----
---
--
-
```

Note that this is very similar to printing a rectangle (cf. Problem 14.6), except that the width of the row decreases with each recursive function call.

Problem 14.8: Printing rightside-up triangles

Copy-and-paste the contract and definition for the `print-n-dashes` *function from Example 4.2.1 into your Definitions Window. Then define a function, called* `print-rightside-up-triangle`*, that satisfies the following contract:*

```
;;  PRINT-RIGHTSIDE-UP-TRIANGLE
;; ------------------------------------
;;  INPUTS:  NUM-ROWS, a non-negative integer
;;           CURR-WIDTH, the width of the current row
;;  OUTPUT:  Nothing
;;  SIDE EFFECT:  Prints a rightside-up triangle in the
;;   Interactions Window consisting of NUM-ROWS rows.
```

Here are some examples illustrating its behavior:

```
> (print-rightside-up-triangle 3 1)
-
--
---

> (print-rightside-up-triangle 5 1)
-
--
---
----
-----
```

Notice that this function is called with `curr-width` *equal to* `1`*, because that's the width of the first row to be printed.*

Problem 14.9

Suppose that `func` *is a function whose output values are within some small non-negative range, say, from 0 to 50. For example, suppose that* `(func 3)` *evaluates to* `25`*. That output value could be represented graphically by a horizontal line containing 25 asterisks. Similarly, if* `(func 4)` *evaluates to* `16`*, then the next line of printing could show 16 asterisks. Your job is to define a function, called* `plotter`*, that plots the output values of a given function over a specified range of inputs. Here's the contract:*

```
;;  PLOTTER
;; -----------------------------------------------------------------
;;  INPUTS:  FUNC, a function that expects a numerical input
;;           FROM, a starting input value (an integer)
;;           TO, an ending input value (an integer)
;;  OUTPUT:  None
;;  SIDE EFFECT:  Displays the output vaules of FUNC for each
;;    input in the range, FROM, FROM+1, FROM+2, ..., TO-2, TO-1, TO.
;;    For each input value, the corresponding output value is
;;    displayed by the appropriate number of asterisks printed on a
;;    single line of the Interactions Window.
```

Here is an example that uses `facty-v1` *from Example 12.1.2:*

```
> (plotter facty-v1 1 4)
*
**
******
************************
```

And here is an example using `abs`*, a built-in function that computes the absolute value of its input. (Notice what happens when* `abs` *is given an input of zero.)*

```
> (plotter abs -3 3)
***
**
*

*
**
***
```

Finally, here's an example where that uses `lambda` *to create a squaring function on the spot, without bothering to give it a name!*

```
> (plotter (lambda (x) (* x x)) 1 5)
*
****
*********
****************
*************************
```

Although you can run the above examples in the Interactions Window, you should also put the corresponding `alt-tester` *expressions in your Definitions Window. Insert additional* `alt-tester` *expressions to demonstrate that your* `plotter` *function works as desired.*

Problem 14.10

⟹ *This problem assumes that you have already defined the* `print-thing-n-times` *function from Problem 14.4. That function can be used to print a line of + signs, or a line of − signs, in the Interactions Window.*

Define a tail-recursive *function called* `fancy-plotter` *that satisfies the following contract:*

```
;;  FANCY-PLOTTER
;; --------------------------------------------------------
;;  INPUTS:  FUNC, a function that takes numerical input
;;           FROM, a starting number (integer)
;;           TO, a stopping number (integer)
;;  OUTPUT:  None
;;  SIDE EFFECTS:  This function plots the function values for FUNC
;;    for each input in the range from FROM to TO.  Each input
;;    value will generate one line of printing in the Interactions
;;    Window.  For example, if (FUNC FROM) ==> 5, then this
;;    function will display a line of five + signs; if (FUNC FROM)
;;    ==> -5, then this function will display a line of five -
;;    signs; if (FUNC FROM) ==> 0, then this function will simply
;;    display a zero.
```

Here are some examples, one of which uses the built-in `sin` *function:*

```
> (fancy-plotter (lambda (x) (* x x x)) -3 3)
---------------------------
--------
-
0
+
++++++++
+++++++++++++++++++++++++++

> (fancy-plotter (lambda (x) (* 20 (sin (/ x 4)))) -20 20)
++++++++++++++++++++
++++++++++++++++++++
++++++++++++++++++++
++++++++++++++++++
++++++++++++++++
++++++++++++
++++++++
+++
---
--------
------------
----------------
-------------------
--------------------
--------------------
-------------------
-----------------
--------------
----------
-----
0
+++++
++++++++++
++++++++++++++
+++++++++++++++++
+++++++++++++++++++
++++++++++++++++++++
++++++++++++++++++++
+++++++++++++++++++
++++++++++++++++
++++++++++++
++++++++
+++
---
--------
------------
----------------
------------------
--------------------
--------------------
--------------------
```

Problem 14.11

For this problem, you will implement a `print-checkerboard` *function that displays a checkerboard pattern in the Interactions Window.*

(a) Define a tail-recursive *function called* `print-checkerboard-acc` *that takes four inputs:* `num-rows`, `num-cols`, `curr-row` *and* `curr-col`. `num-rows` *and* `num-cols` *specify the overall size of the checkerboard;* `curr-row` *and* `curr-col` *specify the location of the next square to be printed.*

When called with appropriate initial values for `curr-row` *and* `curr-col`, *this function should cause a* `num-rows`*-by-*`num-cols` *checkerboard pattern to be printed in the Interactions Window.*

- *The values of* `num-rows` *and* `num-cols` *should not change across the various recursive function calls, but the values of* `curr-row` *and* `curr-col` *will change.*
- *If the current square is somewhere in the middle of the board, this function should print just that one square. It should then let the recursive function call print the rest of the checkerboard. (How should the values of* `curr-row` *and* `curr-col` *be updated in this case?)*
- *If the sum of* `curr-row` *and* `curr-col` *is even, then print one kind of square (e.g.,* `X`*); if their sum is odd, then print the other kind of square (e.g.,* `_`*). (You may use the built-in functions,* `even?` *and* `odd?`*, to test whether a given number is even or odd.)*
- *How do you recognize that you have already finished printing out the entire checkerboard (i.e., you've hit the base case)?*
- *How do you recognize that you have finished printing out the current row? How should the values of* `curr-row` *and* `curr-col` *be updated in that case?*
- *Summary of cases to consider:*
  - ⋆ *You're in the middle of a row.*
  - ⋆ *You've hit the end of a row, but not the last row.*
  - ⋆ *You've hit the end of the last row.*

(b) Define a wrapper *function called* `print-checkerboard` *that takes two inputs,* `num-rows` *and* `num-cols`. *It should cause a* `num-rows`*-by-*`num-cols` *checkerboard pattern to be displayed in the Interactions Window, as illustrated below:*

```
> (print-checkerboard 3 6)
X _ X _ X _
_ X _ X _ X
X _ X _ X _
```

Note that this function should just call the function from part (a) with appropriate inputs. You may wish to use the `alt-tester` *function when writing test cases in the Definitions Window.*

Problem 14.12

The following functions use the built-in `quotient` *and* `remainder` *functions to access the individual digits in the base-ten representation of a number. For the purposes of this problem, the rightmost digit in a number will be considered to be in position zero, the next rightmost digit in position one, and so on. For example, the* `3` *in* `9999399` *will be considered to be in position* `2`*.*

(a) Define a function called `nth-rightmost-digit` *that satisfies the following contract:*

```
;;  NTH-RIGHTMOST-DIGIT
;; -------------------------------------------------------------
;;  INPUTS:  NUM, a non-negative integer
;;           N, a non-negative integer
;;  OUTPUT:  The Nth rightmost digit of NUM, where N=0 refers
;;             to the rightmost digit.
```

Here are some examples of the desired behavior:

```
> (nth-rightmost-digit 92845 0)
5
> (nth-rightmost-digit 92845 2)
8
```

Hint: Consider how dividing a number by ten, using the built-in `quotient` *and* `remainder` *functions, can effectively "peel off" the rightmost digit of the number.*

```
> (quotient 345678 10)
34567
> (remainder 345678 10)
8
```

(b) Define a tail-recursive, accumulator-based function called `num-occurs-acc` *that satisfies the following contract:*

```
;;  NUM-OCCURS-ACC
;; ---------------------------------------------
;;  INPUTS:  DIGIT, an integer from 0 to 9, inclusive
;;           NUM, a non-negative integer
;;           ACC, an accumulator
;;  OUTPUT:  When called with ACC = 0, the output is the number
;;    of occurrences of DIGIT in the decimal repr'n of NUM.
;;  Example:  (num-occurs-acc 3 32123334 0) ==> 4
```

After you have done so, then define the following "wrapper" function:

```
;;  NUM-OCCURS-WR -- wrapper function for NUM-OCCURS-ACC
;; ---------------------------------------------------------
;;  INPUTS:  DIGIT, an integer from 0 to 9, inclusive
;;           NUM, a non-negative integer
;;  OUTPUT:  The number of occurrences of DIGIT in the decimal
;;            representation of NUM.

(define num-occurs-wr
  (lambda (digit num)
    (num-occurs-acc digit num 0)))
```

Here are some examples of the desired behavior:

```
> (num-occurs-wr 3 12312344444443)
3
> (num-occurs-wr 0 100)
2
> (num-occurs-wr 5 1234)
0
```

Hint 1: Use the built-in `quotient` *and* `remainder` *functions. In the context of this problem, consider the following examples:*

```
> (quotient 345678 10)
34567
> (remainder 345678 10)
8
```

So, dividing by ten each time allows you to effectively "peel off" the rightmost digit.

Hint 2: The base case should be when you have exactly one digit left (i.e., when `num` $\leq$ *ten).*

Problem 14.13: Approximating the natural logarithm function

Mathematicians tell us that the natural logarithm *function can be approximated using certain kinds of sums. In particular, for any real number* $x \in (-1, 1]$, *and for any "sufficiently large" positive integer* n, *the value* $\log(1 + x)$ *is well approximated by the following sum:*

$$x \;-\; \frac{x^2}{2} \;+\; \frac{x^3}{3} \;-\; \frac{x^4}{4} \;+\; \ldots \;\pm\; \frac{x^n}{n}$$

For example, the value $\log(1.5)$, *where* $x = 0.5$, *is well approximated by the sum:*

$$0.5 \;-\; \frac{(0.5)^2}{2} \;+\; \frac{(0.5)^3}{3} \;-\; \frac{(0.5)^4}{4}$$

Admittedly, these facts are probably not obvious! But that's okay, since this is not a math class! We can just accept what the mathematicians tell us and go about the business of computing these kinds of sums.

So... for this problem, define an accumulator-based function, called `approx-log-acc` *that satisfies the following contract. Your function should be* tail-recursive. *Also, you may want to use the built-in* `even?` *and* `odd?` *functions, seen previously. The* `power` *function, from Problem 12.1, should be helpful.*

```
;;  APPROX-LOG-ACC
;; ----------------------------------------------------------------
;;  INPUTS:  X, any number such that -1 < X <= 1
;;           FROM, the index of the "current term" in the sum
;;           TO, the index of the "last term" in the sum
;;           ACC, an additive accumulator
;;  OUTPUT:  When called with FROM=1, ACC=0, and TO=N the output is
;;           the sum:  X - X*X/2 + X*X*X/3 - ... (+/-)X^N/N
```

After you have defined `approx-log-acc`*, define the following wrapper function:*

```
(define APPROX-LOG-WR
  (lambda (x to)
    ;; Call the acc-based helper function with FROM=1 and ACC=0
    (approx-log-acc x 1 to 0.0)))
```

Notice that `(approx-log-wr 0.5 4)` *should compute the sum seen earlier that is supposed to be a good approximation of* $\log(1.5)$*. To verify these sorts of examples, you can use Scheme's built-in* `log` *function, but keep in mind that the above sums are good approximations for* $\log(1+x)$*, not for* $\log(x)$*. So, for example, the expression* `(approx-log-wr 0.5 4)` *will evaluate to a good approximation of* $\log(1.5)$*, since* $1 + 0.5 = 1.5$*.*

```
> (approx-log-wr 0.5 10)        ⟵  x = 0.5
0.4054346478174603
> (log 1.5)                     ⟵  1 + x = 1.5
0.4054651081081644
```

Although `approx-log-wr` *will be good at estimating values of* $\log(1+x)$ *for small values of* x*, it doesn't do so well as the value of* x *approaches* 1*. Compare the values returned by* `(approx-log-wr 1 n)` *for various values of* `n` *against the value of* `(log 2)`*. How big does* `n` *have to be before the answer is correct to within 3 decimal places? Be sure to include a variety of* `tester` *expressions in your definitions file to see how well expressions of the form* `(approx-log-wr x n)` *approximate* $\log(1+x)$ *for various values of* `x` *and* `n`*.*

Problem 14.14: Computing geometric sums

This problem concerns the computation of sums such as those shown below:

$$1 \;+\; 10 \;+\; 10^2 \;+\; 10^3 \;=\; 1+10+100+1000 \;=\; 1111$$

$$1 \;+\; 2 \;+\; 2^2 \;+\; 2^3 \;+\; 2^4 \;=\; 1+2+4+8+16 \;=\; 31$$

$$1 \;+\; 3 \;+\; 3^2 \;+\; 3^3 \;=\; 1+3+9+27 \;=\; 40$$

More generally, for any number x *and any non-negative integer* n*, the following expression is called a* geometric sum*:*

$$1 \;+\; x \;+\; x^2 \;+\; x^3 \;+\; \ldots \;+\; x^n$$

where terms of the form x^k *stand for "*x *raised to the* k^{th} *power". Your job is to define a function, called* `geom`*, that takes two inputs,* `x` *and* `n`*, and whose output is the value of the corresponding geometric sum, as shown above.*
Now, there are lots of ways to do this problem. Here, we are going to focus on a way that involves defining an accumulator-based tail-recursive helper function, called `geom-helper-acc`*, that takes the following additional inputs:*

- `k`*, a counter that goes from* 0 *up to* n

- `x-to-the-k`, *a variable that takes on the values,* $1, x, x^2, x^3$, *etc.*
- `acc`, *a variable that accumulates the desired sum*

(When I say that `k` *is a counter "that goes from* 0 *up to* n*", I really mean that the value of the input* `k` *on* successive recursive function calls *increases by one each time.)*
Consider the case where $x = 2$ *and* $n = 4$*. The sum we want to compute is:* $1 + 2 + 2^2 + 2^3 + 2^4$*, which happens to be equal to* 31*. Here are the successive values we want the variables,* `k`, `x-to-the-k` *and* `acc` *to take on during successive recursive function calls:*

`k`	*0*	*1*	*2*	*3*	*4*	*5*	← *We'll stop here, since* $k > 4$
`x-to-the-k`	*1*	*2*	*4*	*8*	*16*	*32*	
`acc`	*0*	*1*	*3*	*7*	*15*	*31*	← *That's the desired answer!!*

As suggested earlier, the value of `k` *increases by one for each successive recursive function call;* `x-to-the-k` *is* multiplied *by* `x` *each time; and* `acc` *accumulates the most recent value of* `x-to-the-k`*. In particular, the value of* `acc` *in one column is the* sum *of the values of* `x-to-the-k` *and* `acc` *from the preceding column:*

$$1 + 0 \Longrightarrow 1; \quad 2 + 1 \Longrightarrow 3; \quad 4 + 3 \Longrightarrow 7; \quad 8 + 7 \Longrightarrow 15; \quad 16 + 15 \Longrightarrow 31.$$

Okay, you are now ready to define the accumulator-based, tail-recursive helper function, `geom-helper-acc`*. It should take the following inputs:* `x`, `n`, `k`, `x-to-the-k` *and* `acc`.

- ⋆ *For the base case... when should this function stop?*
- ⋆ *For the recursive case... make a tail-recursive function call with appropriately adjusted inputs.*

Afterward, you can define `geom` *as a wrapper function that simply calls the above helper function with appropriate initial values for its five inputs. Here's how it should work in the end:*

```
> (geom-sum 10 3)
1111
> (geom-sum 2 4)
31
```

As always, be sure to include contracts for each function you define.

Problem 14.15: Approximating the *arctangent* function

Mathematicians tell us that the arctangent *function can be approximated using sums of the following form:*

$$arctan(x) = 1 - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \ldots \pm \frac{x^n}{n}$$

where n *is any odd positive integer. Define a function, called* `approx-arctan-acc`*, that uses the following inputs to keep track of relevant information over the course of the recursive function calls:*

`x`	*Fixed value*
`from`	*Positive odd integer, numerator of current term*
`sign`	*Alternates between* 1 *and* -1
`curr-power`	*Current power of* x *(computed incrementally)*
`n`	*Fixed value, indicates last term in the sum*
`acc`	*An accumulator*

What initial values should be given to the inputs `from`, `sign`, `curr-power` *and* `acc`*?*